A Survey of GPU-Based Volume Rendering of Unstructured Grids

Cláudio T. Silva [1]
João L. D. Comba [2]
Steven P. Callahan [1]
Fabio F. Bernardon [2]

Abstract:

Real-time rendering of large unstructured meshes is a major research goal in the scientific visualization community. While, for regular grids, texture-based techniques are well-suited for current Graphics Processing Units (GPUs), the steps necessary for rendering unstructured meshes are not so easily mapped to current hardware. This paper reviews volume rendering algorithm and techniques for unstructured grids aimed at exploiting high-performance GPUs. We discuss both the algorithms and their implementation details, including major shortcomings of existing approaches.

Resumo:

A visualização volumétrica de grandes malhas não estruturadas é uma das principais metas da comunidade de visualização científica. Enquanto que em grades regulares o uso de técnicas baseadas em textura são adequadas para as Unidades de Processamento Gráfico (GPUs) atuais, os passos necessários para exibir malhas não estruturas não são diretamente mapeadas para o hardware atual. Este artigo revisa algoritmos e técnicas de visualização volumétrica que exploram GPUs de alta performance. São discutidos tanto os algoritmos como seus detalhes de implementação, incluindo as principais dificuldades das abordagens atuais.

1 Introduction

This paper contains a survey of volume visualization techniques for unstructured volumetric meshes. This manuscript was not designed as a comprehensive survey. Instead, we cover the material that will be presented at our tutorial to be given at the XVII Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI 2005). Our emphasis on unstructured data is unique when compared to other recent surveys, e.g., [7, 31], which are

---

[1]Scientific Computing and Imaging Institute, University of Utah, 50 S Central Campus Dr, Room 3490, Salt Lake City, UT 84112, USA
{csilva@cs.utah.edu, stevec@sci.utah.edu}

[2]Universidade Federal do Rio Grande do Sul, Instituto de Informática, Av. Bento Goncalves, 9500, Campus do Vale, Bloco IV, Prédio 4325, Porto Alegre, RS 91501-970, Brasil
{comba@inf.ufrgs.br, fabiofb@inf.ufrgs.br}

often comprehensive in coverage and have a stronger focus on regular datasets. Our coverage is necessarily biased to the work that we know best, and to that work that we have performed ourselves. Whenever possible, we try to give extra insights on some topics not directly described in the original references, or to point to more general frameworks that can enlighten the material. We hope our discussion of current techniques is useful to other people interested in this area. (We wrote this paper when the current GPUs were represented by the ATI X800 and the NVIDIA 6800 series.)

2   Volume Rendering Techniques

For the visualization of three-dimensional scalar fields, direct volume rendering has emerged as a leading, and often preferred, method. In rendering volumetric data directly, we treat space as composed of semi-transparent material that can emit, transmit, and absorb light, thereby allowing one to "see through" (or see inside) the data [49]. Volume rendering also allows one to render surfaces, and, in fact, by changing the properties of the light emission and absorption, different lighting effects can be achieved.

Volumetric representation leads to different visualization techniques. For instance, regular grids have a well-defined structure that can be explored when designing visualization algorithms. Today, texture-based visualization [29] is central to many different algorithms for regular grids. On the other hand, unstructured grids composed of meshes of tetrahedra are more challenging, and have led to a larger variety of different techniques, e.g., [10, 48, 55, 61, 64, 66, 70]. In this section we review some of the techniques that have been efficiently mapped to GPUs.

2.1   Texture-Based Techniques

In many volumetric applications, data is obtained from a regular sampling approach that leads to a discrete representation of regular grids. In 3D, regular grids are usually stored as a volumetric table containing data from the sampled function (e.g., scalar values, color, etc.) in each cell (usually referred to as a voxel). Generating volume rendering images of regular grids requires sampling the information stored in this table in visibility order (either front-to-back or back-to-front). This requires either computing successive intersections of the volume against a given ray (a ray casting approach) or along parallel planes.

Texture-based visualization is a technique that exploits the texture-based functionality of the graphics hardware to solve the above problem efficiently. Regular grid data is stored directly in the GPU memory as textures. Early graphics boards only supported 2D textures in hardware, which made volumetric visualization only possible by representing volumetric data as stacks of 2D images. Preliminary approaches sample the volume with parallel

Figure 1. Texture-based visualization uses parallel planes orthogonal to the viewing direction to sample the volume.

planes (proxy geometry) that change according to the the viewing parameters. To avoid artifacts, data is replicated in the three directions, increasing memory usage. Avoiding this extra storage requires on-the-fly slice reconstruction at the expense of performance overhead [38]. Another problem that arises here is the dependency on the sampling rate used to produce the regular grid, which can be avoided by using extra slices and trilinear filtering [56].

The advent of 3D texturing capabilities even further improved the ways that the graphics hardware can be used to perform volumetric visualization. Since the volume data is naturally stored as a 3D texture, it suffices to use a proxy geometry consisting of a set of parallel polygons orthogonal to the viewing direction (Figure 1).

The scalar values stored in 3D textures are processed in pairs of successive slices. For each pair of scalar values, lighting calculation accumulate the color contribution of one ray segment using the volume rendering integral. The use of approximations of the volume rendering integral calculations are often useful to speed up this process. The pre-integration approach described in [22] computes an approximation that takes into account the scalar values sampled at two different positions of the ray and the distance between each sample to produce color and opacity values based on a given transfer function. The resulting calculations are stored as a 3D texture accessed by the scalar values (x-axis and y-axis) and distance (z-axis) during rasterization, then combined into the accumulated final color.

The discretization used in the pre-integration might introduce some artifacts because it assumes a linear progression of the scalar values between slices. An adaptive approach is proposed in [57] that tries to overcome this problem with an adaptive sampling rate based on the actual information stored in the volume and the transfer function used.

In [37], several techniques are discussed to reduce the cost of per-fragment operations perfomed during texture-based visualization. Empty-space skipping techniques allow
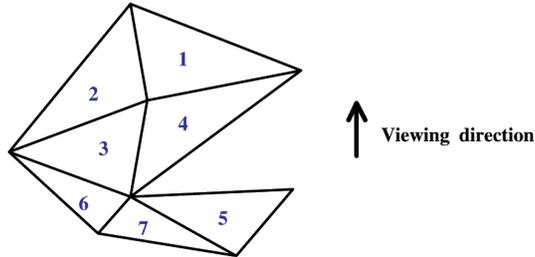
Figure 2. A back-to-front ordering of the cells of a mesh. Note that a visibility ordering can be somewhat unintuitive. For instance, note that cell 3 comes before cell 4 in a correct visibility ordering, although the centroid of cell 4 is farther from the viewer than the centroid of cell 3. This same error could occur if the sort was based on the power distance. This is the reason that simple schemes, such as power distance sorts or centroid-based sorts, may fail from certain viewpoints or viewing directions.

processing to quickly skip entire regions of the volume that do not contain relevant information, which is often the case in sparse datasets. In addition, an early-ray termination allows computation to stop when sufficient opacity values have been accumulated.

2.2 Visibility Ordering

A visibility ordering of a set of objects (see Figure 2), from a given viewpoint, is a total order on the objects such that if object *a* obstructs object *b*, then *b* precedes *a* in the ordering. Such orderings are useful for rendering volumetric data, because it enables efficient use of graphics hardware [61, 70].

In computer graphics, work on visibility ordering was pioneered by Schumacker *et al.* and is later reviewed in [65]. An early solution to computing a visibility order given by Newell, Newell, and Sancha (NNS) [51] continues to be the basis for more recent techniques [64]. The NNS algorithm starts by partially ordering the primitives according to their depth. Then, for each primitive, the algorithm improves the ordering by checking whether other primitives precede it or not. Fuchs, Kedem, and Naylor [26] developed the Binary Space Partitioning tree (*BSP-tree*) — a data structure that represents a hierarchical convex decomposition of a given space (typically, $\mathbb{R}^3$). Since visibility order is essential for volume rendering unstructured grids via cell projection, many techniques have been developed that order the tetrahedra [16, 18, 34, 62, 70].

An intuitive overview of the Meshed Polyhedra Visibility Ordering (MPVO) algorithm [70] is as follows (see Figure 3). First, the adjacency graph for the cells of a given convex
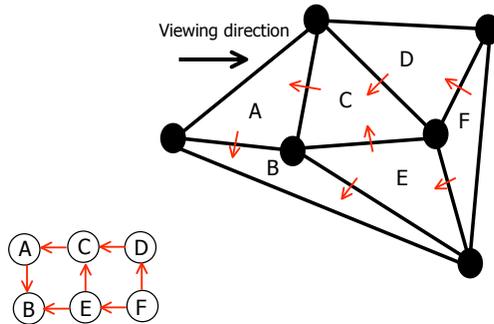
Figure 3. In the top right, we show a set of cells in a convex mesh and their MPVO arrows marked in red. In the bottom left, we show the resulting directed graph. Cell B is a sink cell. To generate the ordering, a topological sort is computed starting from sink cells.

mesh is constructed (for non-convex meshes, see [16–18, 34, 62]). Then, for any specified viewpoint, a visibility ordering can be computed simply by assigning a direction to each edge in the adjacency graph and then performing a topological sort of the graph. The adjacency graph can be reused for each new viewpoint and for each new dataset defined on the same static mesh. The direction assigned to each edge of the adjacency graph is determined by calculating the behind relation for the two cells connected by the edge. Informally, the behind relation is calculated as follows. Each edge corresponds to a facet shared by two cells. That facet defines a plane which in turn defines two half-spaces, each containing one of the two cells. If we represent the behind relation by a directed arc (arrow) through the shared face, then the direction of the arrow is towards the cell whose containing half-space contains the viewpoint. To implement this, the plane equation for the shared face can be evaluated at the viewpoint v. The adjacency graph and the plane equation coefficients can be computed and stored in a preprocessing step. Because of geometric degeneracies, the implementation of MPVO can be complex, and needs to be performed with care.

The methods presented above operate in *object-space*, i.e., they operate on the primitives before rasterization by the graphics hardware [2]. Carpenter [11] proposed the A-buffer — a technique that operates on pixel fragments instead of object fragments. The basic idea is to rasterize all the objects into sets of pixel fragments, then save those fragments in per-pixel linked lists. Each fragment stores its depth, which can be used to sort the lists after all the objects have been rasterized. A nice property of the A-buffer is that the objects can be rasterized in any order, and thus, do not require any object-space ordering. A main shortcoming of the A-buffer is that the memory requirements are substantial. Recently, there have been

Classification

Class 1a	Class 1b	Class 2	Class 3a	Class 3b	Class 4
+ + + -	+ - - -	+ + - -	+ + - 0	+ - - 0	+ + - 0
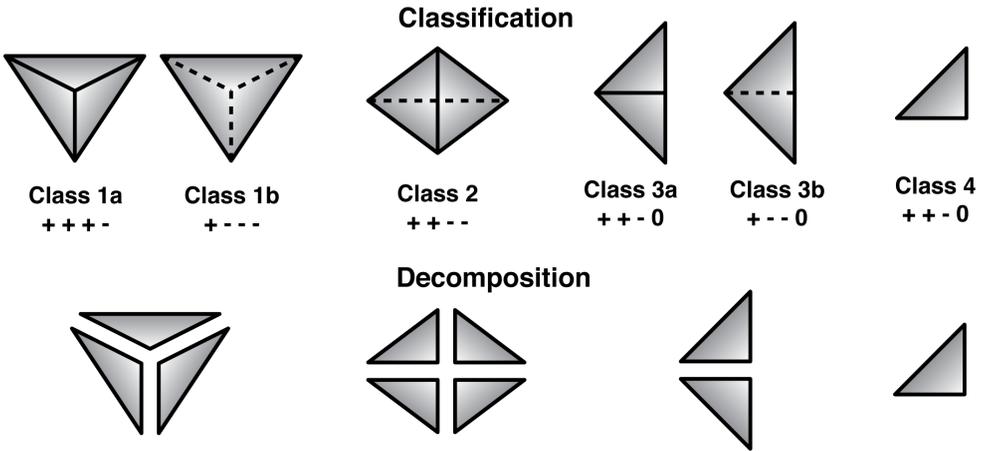
Decomposition

Figure 4. Classification and decomposition of tetrahedra in the Projected Tetrahedra algorithm.

proposals for implementing the A-buffer in hardware [23, 30, 36, 47, 71].

2.3 Splatting

Once an object-space visibility ordering has been computed, one can render a mesh by projecting its sorted cells on the screen one at a time. The leading technique for performing this computation is the Projected Tetrahedra (PT) algorithm [61].

Projection of a volume cell is a task which does not easily map to graphics hardware. Shirley and Tuchman introduced the PT algorithm as a way to approximate a volume cell with a set of partially transparent polygons that can be rendered quickly on graphics hardware. The algorithm works by classifying a cell based on the visibility of the triangle faces with respect to the viewpoint. This classification is done by using a plane equation for each face to determine if the face is visible ('+'), hidden ('-'), or coplanar ('0') with respect to the eye. The cells are then decomposed into one to four triangles based on the classification (see Figure 4). Using this classification, a new vertex may be computed and the color and opacity of the decomposed triangles are linearly interpolated from the ray integral at the thickest point of the tetrahedron. Finally, the decomposed triangles are composited into the framebuffer using the Porter and Duff over operator [54]. Many improvements have been proposed to improve image quality and performance of the original PT algorithm [32, 35, 58, 64, 72].

With the advent of programmable graphics hardware, even more of the processing

burden has been shifted to the GPU. Wylie *et al.* [72] introduce an extension of the PT algorithm that uses vertex shaders to perform the classification and decomposition of the volume cells. This is done by creating a fixed topology for all tetrahedron called the *basis graph* that contains four vertices of the cell with an additional *phantom* vertex. When a cell is projected, the four vertices and an invalid phantom vertex are sent to the GPU as a triangle fan. A vertex program then performs a series of geometric tests and uses the results of these tests to determine the classification of the cell. Using the classification, the vertices are mapped to the basis graph and the phantom vertex is calculated. Once the new geometry is formed, the thickness of the cell is determined at the phantom vertex by linearly interpolating the vertices. Using this thickness, the color and opacity are assigned to the phantom vertex from a lookup table. These modified faces are then rasterized and composited into the framebuffer.

The technique proposed by Wylie *et al.* can be considered as a general way to map geometric computations into GPUs. Since current GPUs can not change the topology of the graphics primitive, the idea is to always send the most general topology. Also, as no data can be saved across vertex primitives, each primitive needs to be send with all the relevant data structures. The actual computation is stored as a type of finite automata that operates differently as it is given a set of different flags. It is quite instructive to study their source code that is available on the web. (See also [52].)

2.4   Ray Casting

Garrity [28] pioneered the use of ray casting for rendering unstructured meshes. His technique was based on exploiting the intrinsic connectivity available on the mesh to optimize ray traversal, by tracking the entrance and exit points of a ray from cell to cell. Because meshes are not necessarily convex, any given ray may get in and out of the mesh multiple times. To avoid expensive search operations during re-entry, Garrity used a hierarchical spatial data structure to store boundary cells.

Bunyk *et al.* [8] proposed a different technique for handling non-convexities in ray casting. Instead of building a hierarchical spatial data structure of the boundary cells, they exploit the discrete nature of the image. And simply determine for each pixel in the screen, the fragments of the front boundary faces that project there (and are stored in front-to-back order). During ray traversal, each time a ray exits the mesh, Bunyk *et al.* use the boundary fragment intersection to determine whether the ray will re-enter the mesh, and where (i.e., the particular cell). This approach turns out to be simpler, faster, and more robust to floating-point calculations.

Weiler *et al.* [66] propose a hardware-based rendering algorithm based on the work of Garrity. They store the mesh and traversal data structures on the GPU using 3D and 2D textures respectively. In their technique, there is no need for information to be send back
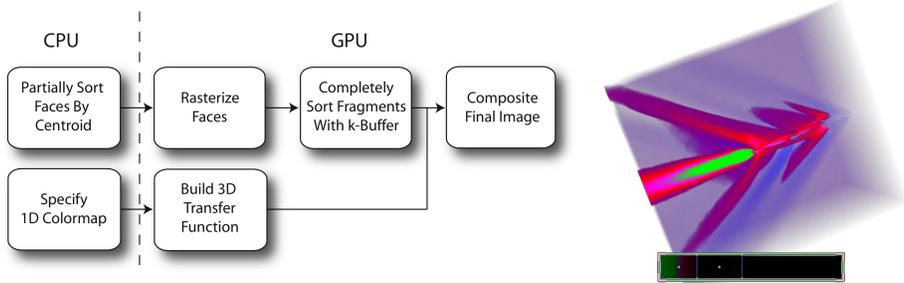
Figure 5. Overview of the HAVS algorithm and volume rendered results. HAVS is a new volume rendering algorithm based on the *k*-buffer that is simpler and more efficient than existing techniques.

and forth between the CPU and GPU, because all computations are performed directly on the GPU. GPU performance currently outpaces the CPU, thus naturally leading to a technique that scales well over time as GPUs get faster. Strictly speaking, their algorithm only handles convex meshes (i.e., it is not able to find the re-entry of rays that leave the mesh temporarily), and they use a technique originally proposed by Williams [70] that calls for the *convexification* of the mesh, and the markings of exterior cells as *imaginary* so they can be ignored during rendering. Unfortunately, convexification of unstructured meshes has many unresolved issues that make it a hard (and currently unsolved) problem. We point the reader to Comba *et al.* [17] for details.

An improved GPU-based ray caster has been proposed by Bernardon *et al.* [5]. In their work, they propose an alternate representation for mesh data in 2D textures that is more compact and efficient, compared to the 3D textures used in the original work. They also use a tile-based subdivision of the screen that allows computation to proceed only at places where it is required, thus reducing fragment processing on the GPU. Finally, their technique does not introduce *imaginary* cells that fill space caused by non-convexities of the mesh. Instead, they use a depth-peeling approach that captures when rays re-enter the mesh, which is much more general and does not require a convexification algorithm. Concurrently with Bernardon *et al.*, Weiler and colleagues developed a similar depth-peeling approach to handling non-convex meshes [67].

2.5 Hardware Assisted Visibility Sorting

Roughly speaking, all of the techniques described above perform sorting *either* in object-space *or* in image-space exclusively. There are also hybrid techniques that sort both in image-space and object-space [1, 24]. Callahan *et al.* [10] proposes a visibility ordering algo-

rithm that works in both image- and object-space (see Figure 5). The object-space sorting is performed on the CPU and results in an approximate order of the geometry. The image-space sorting finalizes the order of the fragments and is done entirely on the GPU using a novel data structure called the *k*-buffer. The original A-buffer [11] stores all incoming fragments in a list, which requires a potentially unbounded amount of memory. The *k*-buffer approach stores only a fixed number of fragments and works by combining the current fragments and discarding some of them as new fragments arrive. This technique reduces the memory requirement and is simple enough to be implemented on existing graphics architectures.

The *k*-buffer is a *fragment stream sorter* that works as follows. For each pixel, the *k*-buffer stores a constant *k* number of entries $\langle f_1, f_2, \ldots, f_k \rangle$. Each entry contains the distance of the fragment from the viewpoint, which is used for sorting the fragments in increasing order for front-to-back compositing and in decreasing order for back-to-front compositing. For front-to-back compositing, each time a new fragment f_{new} is inserted in the *k*-buffer, it dislodges the first entry (f_1). Note that boundary cases need to be handled properly and that f_{new} may be inserted at the beginning of the buffer if it is closer to the viewpoint than all the other fragments or at the end if it is further. A key property of the *k*-buffer is that given a sequence of fragments such that each fragment is within *k* positions from its position in the sorted order, it will output the fragments in the correct order. Thus, with a small *k*, the *k*-buffer can be used to sort a *k*-nearly sorted sequence of *n* fragments in $O(n)$ time.

The availability of the *k*-buffer makes for a very simple rendering algorithm. The Hardware Assisted Visibility Sorting (HAVS) algorithm is built upon the machinery presented above. First, the *faces* of the tetrahedral cells of the unstructured mesh are sorted on the CPU based on the face centroids using the floating point radix sort algorithm. To properly handle boundaries, the vertices are marked whether they are internal or boundary vertices of the mesh. Next, the faces are rasterized by the GPU which completes the sort using the *k*-buffer and composites the accumulated color and opacity into the framebuffer.

3 Isosurface Techniques

One of the most important tools for scientific visualization is isosurface generation. Given a function $f : R^3 \rightarrow R$, an isosurface is the preimage of a given value $y \in R$. In other words, it is the set of all *x* such that $f(x) = y$. Isosurfaces are frequently used for effectively exploring three-dimensional data on a computer. The function *f* is usually represented either by a regular grid of scalar values or by a set of values on a tetrahedral mesh.

The classic algorithm for isosurfacing data laid out on a regular grid is Marching Cubes [41]. Since Marching Cubes touches every cube in the grid, it can be inefficient. There are many published extensions that tackle the original problems [3,6]. For isosurfacing tetrahedral meshes, there is a version of Marching Cubes called Marching Tetrahedra [20], in
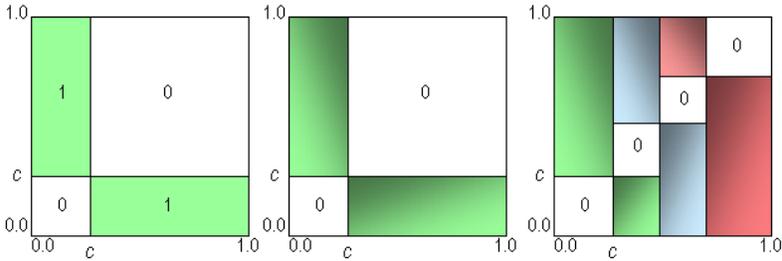
Figure 6. Checkerboard 2D texture used to compute isosurfaces: flat-shaded isosurface (left), smooth-shaded isosurface (center), and multiple isosurfaces (right).

which most extensions of Marching Cubes can be implemented. The fundamental geometric task is computing the isosurface approximation within a tetrahedra, and the various ways of computing plane-tetrahedra intersections leads to the different algorithms in the literature.

A challenging aspect of these techniques is that a change of isovalue incurs on a re-computation of the entire isosurface. This new geometry data must be computed on the CPU and then sent to the graphics processor. The CPU and GPU communicate through a relatively narrow bus, creating a significant bottleneck. Since graphics hardware is becoming faster at a much higher rate than general-purpose processors, a desirable option is to push as much work as possible to the graphics side (GPU).

Early work on this area explored graphics hardware functionality without the power of programmable shaders. In [69] it is shown how plane-tetrahedra intersections can be imple-mented using rasterization hardware with alpha-test functionality and a stencil buffer. Back and front faces of a given cell are drawn in this order with an interpolated scalar value stored at the alpha component, with the appropriate alpha-test configured to reject fragments less than the desired isovalue. The combination of the results obtained with the back and front faces is obtained using a XOR operation and the stencil buffer. The resulting algorithm runs in two passes (the first step sets the stencil buffer) and on average renders 5 to 7 triangles. A revised version of this work is discussed in [58].

In [58], an alternate solution is proposed that renders fewer faces per tetrahedron. The main idea is to use the decomposition step of the PT algorithm to create smaller tetrehedra instead of triangles. The projections of back and front faces of each smaller tetrahedra lead to two simple cases, either a degenerate triangle or the same triangle. The coincidence of this projection allows the intersection computation to proceed using 2D texturing hardware by associating a checkerboard texture. Extracting the isosurface is done by sampling the 2D texture with texture coordinates corresponding to the front and back face scalar values.
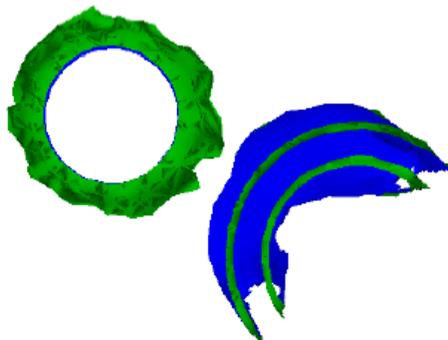
Figure 7. Isosurfaces of the SPX dataset. Front(back) faces are shown in green(blue).

Extensions to handle smooth and multiple isosurfaces require storing shading values in the 2D texture, as well as further partitioning of the texture domain (Figure 6). Figure 7 illustrates the result obtained using the texture-encoded algorithm (a modified version of the HAVS algorithm was used to generate this figure).

Although the intrinsic computation of isosurfaces differ from the visibility computations used for volume rendering, the mapping of these computations onto the GPU using programmable hardware shares in most cases the same underlying framework. For instance, the framework proposed in Wylie *et al.* [72] was revised in Pascucci [52], and represents one of the first solutions to explore programmable hardware to compute isosurfaces. The proposed solution consists of drawing a single quadrilateral for each tetrahedron, which corresponds to the most general topology an isosurface polygon can have inside a tetrahedra. The computation of the isosurface happens inside a vertex shader, where each vertex of the above quadrilateral is repositioned to the exact places the intersections occur, leading to degenerate polygons in some cases. The information required to perform these intersections (tetrahedron edges, endpoints and all possible configurations of marching tetrahedra) are stored as look-up tables in the vertex shader attributes.

One of the reasons that initial solutions used vertex shaders is due to the fact that it was the first part of the graphics pipeline to become programmable. With the advent of fragment shaders, and more general texture formats that could represent floating-point values (among other features), this problem could then be revised to use this new functionality. Since computation with fragment shaders happens at a greater frequency than with vertex shaders, graphics boards have more parallel units in the fragment shader, which represents a greater computational power. In addition, one of the problems with the solution using vertex shaders is that the computed isosurface quadrilaterals were not easily obtained unless each vertex
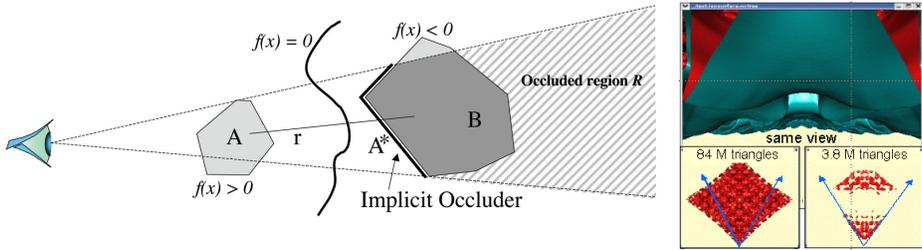
Figure 8. (a) The fundamental idea in *Implicit Occluders* is to exploit the continuity of a scalar field $f(x)$ to define regions of occlusion. In general, any *ray* traveling from a region that is above (or resp. below) the isosurface threshold to one that is below (above) has to intersect the isosurface, thus implicitly (i.e., without the need for computing the surface) defining regions of occlusion. (b) Rendering savings with Implicit Occluders: only 4.5% of the isosurface is rendered.

position was written into a render target and further copied to CPU memory, thus reducing the efficiency of post-processing of isosurface data.

In [33], a solution to compute isosurfaces using fragment shaders is proposed that allows results to be reused with render-to-vertex-arrays functionality. The representation of mesh information in textures required for isosurface computation follows the framework outlined in [66], but with a more compact representation that reduces the memory footprint. In addition, marching tetrahedra tables are stored as textures. A single quadrilateral per tetrahedra is drawn, and inside the fragment shader, each different fragment extracts isosurfaces for a given tetrahedra, producing up to four vertices that are written into four targets using the multiple-render-targets (MRTs) feature. The resulting targets are later bound as vertex arrays and rendered to produce the desired result. The main limitations of this technique are the amount of memory available in the graphics card and the speed of CPU-GPU memory transfers.

Isosurfaces tend to exhibit a large amount of self-occlusion, and extraction of the entire isosurface in a view-independent manner will likely generate occluded elements that make no difference to the final image. When occlusion is taken into account, large parts of the data might not need traversal. Typically, some portions of the isosurface are computed and then used to determine occlusion. A general approach is the use of *implicit occluders* [53], which explore more general scalar field information to determine from-point occlusions *prior* to the isosurface computation. The idea is to exploit the continuity of the underlying scalar field to speed up the computation and rendering of isosurfaces by performing visibility computations (see Figure 8).

Point-based isosurface extraction algorithms dates back to [15] where the Dividing Cubes approach is described, that simply uses a view-dependent approach to divide the model into grid cells until sub-pixel resolution in screen space is achieved. The use of view-dependent isosurface computation is described in [40], which handles larger datasets by considering point primitives to represent smaller triangles. The central idea to the algorithm is a front-to-back traversal of an octree that allows two kinds of pruning to happen. Value-based prunning is responsible for discarding nodes that contain values outside the isosurface limit, which requires storing minimum and maximum values at octree nodes. Visibility pruning allows entire nodes to be discarded based on the fact that the bounding box of a given node may not contribute to the final image. A fragment program is used to compute robust normal positions on the extracted isosurfaces by accessing the positions in world coordinates of the pixel and its neighboring pixels.

4   Advanced Techniques

Through the use of GPUs, rendering large unstructured grids has become a reality. However, since the size of the data is still increasing faster than they can be rendered, advanced techniques need to be employed to retain interactivity.

4.1   Level-of-Detail Techniques

There are several approaches to rendering large unstructured grids. One way to speed up the rendering is to statically simplify the mesh in a preprocessing step to a mesh small enough for the volume renderer to handle [12, 13, 27, 63]. However, this approach only provides a static approximation of the original mesh and does not allow dynamic changes to the level of detail. Multiresolution or level-of-detail (LOD) approaches adapt the amount of data to be rendered to the capabilities of the hardware being used. Dynamic LOD approaches adjust the LOD on a continual basis, and have been shown to be useful by Museth and Lombeyda [50] even if the images generated are more limited than volume rendering.

Cignoni *et al.* [14] propose a technique based on creating a progressive hierarchy of tetrahedra that is stored in a multi-triangulation data structure [25] that is dynamically updated to achieve interactive results. Their multi-triangulation model is created by encoding all the edge-collapses operations that are required to reduce the number of tetrahedra to a base mesh. The mesh is selectively refined to a given LOD using selection heuristics.

Another approach introduced by Leven *et al.* [39] converts the unstructured grid into a hierarchy of regular grids that are stored in an octree. To mitigate the explosion in data size that results from resampling the unstructured grid with a structured grid, the algorithm manages the resampling, filtering, and rendering of the data out-of-core. By resampling the

data in this way, the rendering can be performed using texture-based LOD techniques for regular grids.

A more recent algorithm by Callahan *et al.* [9] takes a much simpler approach by sparsely sampling the primitives that are rendered [19]. Reducing the data size by resampling the domain using a series of edge-collapses can be considered *domain-based* simplification. The authors introduce an alternative approach that removes geometric primitives from the original mesh called *sample-based* simplification. Based on the HAVS volume rendering system described in Section 2.5, which operates on the triangle faces of the tetrahedral mesh; the algorithm keeps a list of the triangle indices sorted by importance. During interaction, the algorithm draws the boundary triangles in addition to a dynamically adjusted number of internal triangles. Several hueristics are explored to determine the importance of a triangle in a preprocessing step.

4.2   Time-Varying and Dynamic Data

Visualizing large datasets interactively is further complicated when the data changes dynamically. Scientific simulation is often interested in measuring or computing the changes of a dataset over time. This can be a simple change of field values on a static mesh, or a combination of changes in the topology and field values. Significant work has been done for structured grids and can be reviewed in surveys by Ma [44] and Ma and Lum [45]. These methods optimize the rendering time by exploring spatial data structures and temporal compression [21, 42, 43, 46, 59, 60, 68, 73].

Recent work by Bernardon *et al.* [4] extend some of the ideas used in the structured case to the unstructured case of time-varying scalar fields. Due to the enormity of data needed to represent a time-varying unstructured grid, compression techniques are essential. The authors extend the vector quantization approach used by [59] to encode the scalar field based on temporal coherence in a hierarchical data structure. The algorithm then dynamically decodes the scalar field for each time step. The authors show that this algorithm can be integrated into a hardware-assisted ray caster [5](see Section 2.4) by decoding each time step and updating the texture containing the scalar values using a fragment shader. Similarly, the decoding can update scalar values for each time step on the HAVS system of Callahan *et al.* [10](see Section 2.5) directly on the CPU. The resulting visualizations incur very little performance overhead to rendering the data statically.

To our knowledge, the more difficult case of time-varying topology has not been addressed for unstructured grids. Currently, these meshes are rendered in a brute force manner. This remains an open topic of research.

5   Discussion

The most popular technique for rendering unstructured tetrahedral meshes is a combination of using the Projected Tetrahedra (PT) technique of Shirley and Tuchman [61] with the MPVO with Non-Convexities (MPVONC) sorting algorithm proposed by Williams [70]. Notwithstanding the fact that the MPVONC sorting is not exact, thus leading to certain visual artifacts, this technique is fast and for most applications "accurate enough". PT also has a number of accuracy and implementation issues that lead to incorrect images. Recently, Kraus *et al.* [35] show how to addresses a number of the limitations of the PT algorithm, in particular, how to implement perspective projection correctly and how to improve the accuracy of compositing on existing GPUs. It is *conceptually* simple to extend MPVO into an exact one by using a strict object-space visibility order to be computed [16, 62]. In practice, it appears to be much more stable (and simple) to guarantee correct visibility order for a particular *image-resolution* using the Scanning Exact MPVO (SXMPVO) algorithm [18]. Overall, most existing codes do not generate strictly correct images, and often are limited to parallel projections because of implementation limitations. Writing a complete system is very complex, and essentially all implementations of these algorithms can be traced back to the original implementations by the authors. A major shortcoming is that the geometric computations required for PT and visibility ordering are hard to implement robustly.

On the surface, GPU-based ray casting [5, 66, 67] appears to sidestep many of the limitations of visibility orders and PT, in particular, the explicit need for computing a visibility order and many of the accuracy issues of PT. Unfortunately, in practice, there are also accuracy limitations of the ray-triangle intersections that appear to lower the image quality of ray casting systems. Also, the need to have all the data in the GPU limits the size of the datasets that can be rendered. (This is not a problem for PT, which simply streams the data from the CPU memory into the GPU.)

Having implemented and tested all these algorithms, it is our opinion that the HAVS algorithm [10] provides the best compromise in terms of implementation complexity, robustness, and image quality. Although it is not *guaranteed* to generate correct images, our HAVS-based system appears to generate better images than all the other techniques we tried. Also, the implementation is very simple and robust. The main shortcoming of HAVS is the requirement of the k-buffer, which relies on advanced (read/write) framebuffer access, a hardware feature that, strictly speaking, is unstable. (But it works fine on current hardware of both ATI and NVIDIA, the two leading manufacturers.) Although there is no reason this feature will not be available on future generations of graphics hardware, there is also no guarantee.

Up to now, we have only addressed the rendering of single-resolution static datasets. Both PT/VO (visibility ordering) and ray casting requires both connectivity information (i.e., which cells are neighbors) and geometric information (i.e., normals to the faces of the cell

complex). Any change in geometry or topology of the mesh requires that at least part of the computation usually done in a preprocessing step be performed again and again. This makes these algorithms hard to use in dynamic settings, which are required not only for time-varying datasets, but also for implementing LOD techniques. With these approaches, it is not feasible to update all the necessary data on a frame-to-frame basis.

We note that HAVS constructs the information necessary for rendering directly on the GPU and does not need any connectivity information. This new class of algorithm does not require any preprocessing, and can, in principle, use a completely different set of data for each frame. Thus, it can potentially handle dynamic geometry.

We end this paper by reiterating that visualization of dynamic unstructured meshes is an area that has been virtually untouched in the literature and lies almost completely in future and ongoing work (see Section 4.2).

6   Acknowledgements

References

[1] T. Aila, V. Miettinen, and P. Nordlund. Delay streams for graphics hardware. *ACM Transactions on Graphics*, 22(3):792–800, July 2003.

[2] ATI. Radeon 9500/9600/9700/9800 OpenGL programming and optimization guide, 2003. http://www.ati.com.

[3] C. L. Bajaj, V. Pascucci, and D. R. Schikore. Fast isocontouring for improved interactivity. In *Proceedings of ACM/IEEE Volume Visualization Symposium 1996*, 1996.

[4] F. F. Bernardon, S. P. Callahan, J. L. D. Comba, and C. T. Silva. Volume rendering of time-varying scalar fields on unstructured meshes. Technical Report UUSCI-2005-006, SCI Institute, 2005.

[5] F. F. Bernardon, C. A. Pagot, J. L. D. Comba, and C. T. Silva. GPU-based tiled ray casting using depth peeling. *Journal of Graphics Tools*, to appear. Also available as SCI Institute Technical Report UUSCI-2004-006.

[6] J. Bloomenthal. *Graphics Gems IV*, chapter An Implicit Surface Polygonizer, pages 324–349. Academic Press, 1994.

[7] K. Brodlie and J. Wood. Recent advances in volume visualization. *Computer Graphics Forum*, 20(2):125–148, 2001.

[8] P. Bunyk, A. Kaufman, and C. T. Silva. Simple, fast, and robust ray casting of irregular grids. In *Proceedings of Dagstuhl '97*, pages 30–36, 2000.

[9] S. P. Callahan, J. L. D. Comba, P. Shirley, and C. T. Silva. Interactive rendering of large unstructured grids using dynamic level-of-detail. In *Proceedings of IEEE Visualization 2005*, 2005. to appear.

[10] S. P. Callahan, M. Ikits, J. L. D. Comba, and C. T. Silva. Hardware-assisted visibility ordering for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295, 2005.

[11] L. Carpenter. The A-buffer, an antialiased hidden surface method. In *Computer Graphics (Proceedings of ACM SIGGRAPH)*, volume 18, pages 103–108, July 1984.

[12] Y.-J. Chiang and X. Lu. Progressive simplification of tetrahedral meshes preserving all isosurface topologies. *Computer Graphics Forum*, 22(3):493–504, 2003.

[13] P. Chopra and J. Meyer. Tetfusion: an algorithm for rapid tetrahedral mesh simplification. In *Proceedings of IEEE Visualization 2002*, pages 133–140, 2002.

[14] P. Cignoni, L. D. Floriani, P. Magillo, E. Puppo, and R. Scopigno. Selective refinement queries for volume visualization of unstructured tetrahedral meshes. *IEEE Transactions on Visualization and Computer Graphics*, 10(1):29–45, 2004.

[15] H. Cline, W. Lorensen, S. Ludke, C. Crawford, and B. Teeter. Two algorithms for the three-dimensional reconstruction of tomograms. *Medical Physics*, 15(3):320–327, 1988.

[16] J. L. D. Comba, J. T. Klosowski, N. Max, J. S. B. Mitchell, C. T. Silva, and P. L. Williams. Fast polyhedral cell sorting for interactive rendering of unstructured grids. *Computer Graphics Forum*, 18(3):369–376, Sept. 1999.

[17] J. L. D. Comba, J. S. B. Mitchell, and C. T. Silva. On the convexification of unstructured grids from a scientific visualization perspective. In G.-P. Bonneau, T. Ertl, and G. M. Nielson, editors, *Scientific Visualization: Extracting Information and Knowledge from Scientific Datasets*. Springer-Verlag, 2005.

[18] R. Cook, N. Max, C. T. Silva, and P. Williams. Efficient, exact visibility ordering of unstructured meshes. *IEEE Transactions on Visualization and Computer Graphics*, 10(6):695–707, 2004.

[19] J. Danskin and P. Hanrahan. Fast algorithms for volume ray tracing. In *Proceedings of Workshop on Volume Visualization 1992*, pages 91–98, 1992.

[20] A. Doi and A. Koide. An efficient method of triangulating equivalued surfaces by using tetrahedral cells. *IEICE Transactions Communication, Elec. Info. Syst*, E74(1):214–224, January 1991.

[21] D. Ellsworth, L.-J. Chiang, and H.-W. Shen. Accelerating time-varying hardware volume rendering using tsp trees and color-based error metrics. In *Proceedings of 2000 Volume Visualization Symposium*, pages 119–128, 2000.

[22] K. Engel, M. Kraus, and T. Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *Proceedings of the EG/SIGGRAPH Workshop on Graphics hardware*, pages 9–16, 2001.

[23] C. Everitt. Interactive order-independent transparency. White paper, NVIDIA Corporation, 1999.

[24] R. Farias, J. Mitchell, and C. T. Silva. ZSWEEP: An efficient and exact projection algorithm for unstructured volume rendering. In *Proceedings of IEEE Volume Visualization and Graphics Symposium*, pages 91–99, 2000.

[25] L. D. Floriani, P. Magillo, and E. Puppo. Efficient implementation of multi-triangulations. In *Proceedings of IEEE Visualization '98*, pages 43–50, 1998.

[26] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. In *Computer Graphics (Proceedings of ACM SIGGRAPH)*, volume 14, pages 124–133, July 1980.

[27] M. Garland and Y. Zhou. Quadric-based simplification in any dimension. *ACM Transactions on Graphics*, 24(2), Apr. 2005.

[28] M. P. Garrity. Raytracing irregular volume data. *Computer Graphics (San Diego Workshop on Volume Visualization)*, 24(5):35–40, Nov. 1990.

[29] M. Ikits, J. Kniss, A. Lefohn, and C. Hansen. *GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics*, chapter Volume Rendering Techniques, pages 667–692. Addison Wesley, 2004.

[30] N. P. Jouppi and C.-F. Chang. Z3: an economical hardware technique for high-quality antialiasing and transparency. In *Procceedings of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 85–93, Aug. 1999.

[31] A. Kaufman. *Encyclopedia of Electrical and Electronics Engineering*, chapter Volume Visualization. Wiley Publishing, 1997.

[32] D. King, C. Wittenbrink, and H. Wolters. An architecture for interactive tetrahedral volume rendering. In *Proceedings of IEEE TVCG/Eurographics International Workshop on Volume Graphics*, pages 101–110, 2001.

[33] T. Klein, S. Stegmaier, and T. Ertl. Hardware-accelerated Reconstruction of Polygonal Isosurface Representations on Unstructured Grids. In *Proceedings of Pacific Graphics '04*, pages 186–195, 2004.

[34] M. Kraus and T. Ertl. Cell-projection of cyclic meshes. In *Proceedings of IEEE Visualization 2001*, pages 215–222, Oct. 2001.

[35] M. Kraus, W. Qiao, and D. S. Ebert. Projecting tetrahedra without rendering artifacts. In *Proceedings of IEEE Visualization 2004*, pages 27–34, 2004.

[36] S. Krishnan, C. T. Silva, and B. Wei. A hardware-assisted visibility-ordering algorithm with applications to volume rendering of unstructured grids. In *Procceedings of EG/IEEE TCVG Symposium on Visualization VisSym 2001*, pages 233–242, 2001.

[37] J. Krueger and R. Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings IEEE Visualization 2003*, 2003.

[38] A. E. Lefohn, J. M. Kniss, C. D. Hansen, and R. T. Whitaker. A streaming narrow-band algorithm: Interactive computation and visualization of level sets. *IEEE Transactions on Visualization and Computer Graphics*, 10(4):422–433, July/Aug. 2004.

[39] J. Leven, J. Corso, J. D. Cohen, and S. Kumar. Interactive visualization of unstructured grids using hierarchical 3d textures. In *Proceedings of IEEE Symposium on Volume Visualization and Graphics*, pages 37–44, 2002.

[40] Y. Livnat and X. Tricoche. Interactive point based isosurface extraction. In *Proceedings of IEEE Visualization 2004*, pages 457–464, 2004.

[41] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of ACM SIGGRAPH*, pages 163–169, 1987.

[42] E. Lum, K.-L. Ma, and J. Clyne. Texture hardware assisted rendering of time-varying volume data. In *Proceedings of IEEE Visualization 2001*, pages 263–270, 2001.

[43] E. Lum, K.-L. Ma, and J. Clyne. A hardware-assisted scalable solution of interactive volume rendering of time-varying data. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):286–301, 2002.

[44] K.-L. Ma. Visualizing time-varying volume data. *Computing in Science & Engineering*, 5(2):34–42, 2003.

[45] K.-L. Ma and E. Lum. Techniques for visualizing time-varying volume data. In C. D. Hansen and C. Johnson, editors, *Visualization Handbook*. Academic Press, 2004.

[46] K.-L. Ma and H.-W. Shen. Compression and accelerated rendering of time-varying volume data. In *Proceedings of 2000 International Computer Symposium – Workshop on Computer Graphics and Virtual Reality*, pages 82–89, 2000.

[47] A. Mammen. Transparency and antialiasing algorithms implemented with the virtual pixel maps technique. *IEEE Comput. Graph. Appl.*, 9(4):43–55, 1989.

[48] N. Max, P. Hanrahan, and R. Crawfis. Area and volume coherence for efficient visualization of 3D scalar functions. *ACM SIGGRAPH Comput. Graph.*, 24(5):27–33, 1990.

[49] N. L. Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[50] K. Museth and S. Lombeyda. Tetsplat: Real-time rendering and volume clipping of large unstructured tetrahedral meshes. In *Proceedings of IEEE Visualization 2004*, pages 433–440, 2004.

[51] M. Newell, R. Newell, and T. Sancha. A solution to the hidden surface problem. In *Proceedings of ACM Annual Conference*, pages 443–450, 1972.

[52] V. Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. In *Proceedings of IEEE TVCG Symposium on Visualization*, pages 293–300, 2004.

[53] S. Pesco, P. Lindstrom, V. Pascucci, and C. T. Silva. Implicit occluders. In *IEEE/SIGGRAPH Symposium on Volume Visualization*, pages 47–54, 2004.

[54] T. Porter and T. Duff. Compositing digital images. *Computer Graphics*, 18(4):253–260, 1984.

[55] D. M. Reed, R. Yagel, A. Law, P.-W. Shin, and N. Shareef. Hardware assisted volume rendering of unstructured grids by incremental slicing. In *Proceedings of the 1996 Symposium on Volume Visualization*, 1996.

[56] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization. In *EG/SIGGRAPH Workshop on Graphics Hardware '00*, pages 109–118, 2000.

[57] S. Roettger, S. Guthe, D. Weiskopf, and T. Ertl. Smart Hardware-Accelerated Volume Rendering. In *Procceedings of EG/IEEE TCVG Symposium on Visualization VisSym 2003*, pages 231–238, 2003.

[58] S. Roettger, M. Kraus, and T. Ertl. Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proceedings of IEEE Visualization*, pages 109–116, Oct. 2000.

[59] J. Schneider and R. Westermann. Compression domain volume rendering. In *Proceedings of IEEE Visualization*, 2003.

[60] H.-W. Shen, L.-J. Chiang, and K.-L. Ma. A fast volume rendering algorithm for time-varying field using a time-space partitioning (tsp) tree. In *Proceedings of IEEE Visualization 1999*, pages 371–377, 1999.

[61] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. *Proceedings of San Diego Workshop on Volume Visualization*, 24(5):63–70, Nov. 1990.

[62] C. T. Silva, J. S. Mitchell, and P. L. Williams. An exact interactive time visibility ordering algorithm for polyhedral cell complexes. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 87–94, Oct. 1998.

[63] O. G. Staadt and M. H. Gross. Progressive tetrahedralizations. In *Proceedings of IEEE Visualization 1998*, pages 397–402, 1998.

[64] C. Stein, B. Becker, and N. Max. Sorting and hardware assisted rendering for volume visualization. In *Proceedings of IEEE Symposium on Volume Visualization*, pages 83–89, Oct. 1994.

[65] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys*, 6(1):1–55, Mar. 1974.

[66] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based ray casting for tetrahedral meshes. In *Proceedings of IEEE Visualization 2003*, pages 333–340, Oct. 2003.

[67] M. Weiler, P. N. Mallón, M. Kraus, and T. Ertl. Texture-Encoded Tetrahedral Strips. In *Proceedings of Symposium on Volume Visualization 2004*, pages 71–78. IEEE, 2004.

[68] Westermann. Compression time rendering of time-resolved volume data. In *Proceedings of IEEE Visualization 1995*, pages 168–174, 1995.

[69] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proceedings of SIGGRAPH '98*, pages 169–177, 1998.

[70] P. L. Williams. Visibility-ordering meshed polyhedra. *ACM Transactions on Graphics*, 11(2):103–126, Apr. 1992.

[71] C. Wittenbrink. R-Buffer: A pointerless a-buffer hardware architecture. In *Proceedings of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 73–80, 2001.

[72] B. Wylie, K. Moreland, L. A. Fisk, and P. Crossno. Tetrahedral projection using vertex shaders. In *Proceedings of IEEE/ACM Symposium on Volume Graphics and Visualization*, pages 7–12, 2002.

[73] H. Yu, K.-L. Ma, and J. Welling. I/O strategies for parallel rendering of large time-varying volume data. In *Eurographics/ACM SIGGRAPH Symposiumm Proceedings of Parallel Graphics and Visualization 2004*, pages 31–40, 2004.